Algoritmi e strutture dati

Koci Erik

April 20, 2021

1 Complessità algoritmi

1. **Costo:** si riferisce al costo di un singolo algoritmo
2. **Complessità:** si riferisce a più risoluzioni di un algoritmo

il costo di un blocco **if-then-else** è $O(max\{f(n), g(n), h(n)\})$ cioè **O(1)**.

1.1 Ordini di grandezza

1. $\Theta(f(n))$ se **cresce tanto quanto f**
2. $O(f(n))$ **se la crescita è minore o uguale a f**
3. $\Omega(f(n))$ **se la crescita è maggiore o uguale a f**

1.2 Esercizi

- $1325n^2 + 12n + 1 = \theta(n^3)$ FALSO
- $76n^3 == (n^3)$ VERO
- $n^2 logn = O(n^2)$ FALSO
- $3^N = O(2^N)$ FALSO
- $1^n = O(2^{\frac{n}{2}})$ FALSO
- $2^N + 100 = O(2^N)$ VERO

- $n = O(nlogn)$ VERO
- $n^2 = (nlogn$ FALSO
- $log(n^2) = \Theta(logn)$ VERO
- $(n+1)/2 = \Theta(n)$ VERO
- $\frac{(n+1)*n}{2} = \Theta(n^2)$ VERO

1.3 Analisi casi

1. **Caso pessimo**
$$T\text{WORST}(n) = maxT(I)$$
2. **Caso ottimo**
$$T\text{BEST}(n) = minT(I)$$
3. **Caso medio**
$$T\text{AVG}(n) = \sum T(I)P(I)$$

1.4 Algoritmi ordinamento

Selection sort: scorre tutti gli elementi degli array e si cerca il valore più piccolo scambiando i due valori. Il costo è lineare con il numero di elementi da considerare:

$$T(n) = \Theta(n^2)$$

il costo è $\Theta(n^2)$ perchè è presente una funzione min che ogni volta controlla se il numero è il minore. Le chiamate a **min** contribuiscono a n^2 mentre il resto combacia con n cioè $n^2 + n$; n viene assorbito.

Ricerca binaria (ricorsiva): per utilizzare questo algoritmo devo avere un array ordinato. Cerco il valore andando a verificare sempre nella metà dove mi aspetto che sia presente.

$$T(n) = 1 \ \ se\ n = 0$$

$$T(n) = T(n/2) + 1$$

equazione di ricorrenza: ci aiuta a calcolare il costo analizzando una singola ricorsione.

1.5 metodo dell'iterazione:

consiste nello sviluppare l'equazione di ricorrenza, per intuirne la soluzione.

$$T(n) = c_1 + c_2 * log(n) = \Theta(log(n))$$

E' presente il **logaritmo** perchè ogni volta devo **dimezzare** il tutto in base al numero di elementi. c_1 perchè devo eseguire le istruzioni la prima volta.

dimostrare che $T(n) = O(n)$

$$T(n = 1)\ n == 1$$

$$T(T([n/2]) + n\ \ n > 1$$

1.6 Metodo della sostituzione:

consiste facendo una dimostrazione per induzione. quindi parto dal valore base che è n (esempio 1) e dimostriamo che vale anche per un n più grande.

caso base:

$$T(1) = 1 \leq cx1$$

induzione:

$$T(n) = T([n/2]) + n$$
$$\leq c[n/2] + n \quad (ipotesi\, induttiva)$$
$$\leq cn/2 + n = \frac{cn + 2n}{2} = (c/2 + 1)n \leq cn$$

1.7 Master Theorem:

Si consideri la seguente equazione di ricorrenza:

$$T(n) = d \ \ se \ n = 1$$
$$T(n) = aT(n/b) + cn^{\beta} \ \ se \ n > 1$$

e sia:

$$\alpha = \frac{log(a)}{log(b)}$$

a numero di chiamate ricorsive
b mi dice come partiziono il mio input
questi due valori mi danno α.
β mi dice l'esponente che avevo.

L'equazione di ricorrenza ha la seguente soluzione:

1. $T(n) = \Theta(n^{\alpha})$ se $\alpha > \beta$
2. $T(n) = \Theta(n^{\alpha} * log(n))$ se $\alpha = \beta$
3. $T(n) = \Theta(n^{\beta})$ se $\alpha < \beta$

Il teorema fondamentale **non** si può applicare ad algoritmi ricorsivi che non effettuano **partizioni bilanciate**.
ad esempio non può essere applicato nella risoluzione di fibonacci ricorsivo.

Esempio:

$$T(n) = 1 \, n <= 1$$
$$T(n1) + T(n2) + 1 \, n > 2$$

Se le **partizioni sono bilanciate** conviene utilizzare il **Master Theorem**.

partizione bilanciate: quando facciamo chiamate ricorsive prendo il mio input suddividendolo in parti n/b. Fibonacci non è bilancianto perche abbiamo due chiamate ricorsive diverse.

L'analisi ammortizzata: studia il costo medio di una sequenza di operazioni.

Sia **T(n, k)** il tempo totale richiesto da un algoritmo, nel caso pessimo, per effettuare k operazioni su istanze di lunghezza n. Definiamo il **costo ammortizzato** su una sequenza di k operazioni come:

$$T_\alpha(n) = \frac{T(n,k)}{k}$$

1.8 Algoritmi di visita degli alberi

Esistono due tipologie di visita:

- In profondità (pre-ordine, in-rodine, post-ordine)
- In ampiezza

Nella visita **pre-ordine** si parte visitando il nodo della radice per poi passare a visitare tutto il nodo sinistro, risalendo poi andando verso destra.

Nella visita **in-ordine** si parte a visitare dal ramo più in basso a sinistra risalendo per poi andare verso destra.

Nella visita **post-ordine** vengono prima visitati i nodi più in profondità partendo sempre da sinistra verso destra per poi risalire.

Nella visita per **ampiezza** si analizza l'albero a livelli, partendo dalla radice.

1.9 Alberi AVL

Un albero AVL è un albero di ricerca (quasi) bilanciato. Questo albero supporta le operazioni di $insert(), delete(), search()$ con costo $O(logn)$ nel **caso pessimo**.

1.9.1 Fattore di bilanciamento

Il fattore di bilanciamento $\beta(v)$ di un novo v è dato dalla differenza tra l'altezza del sottoalbero sinistro e del sottoalbero destro di v:

$$\beta(v) = altezza(v.left) - altezza(v.right)$$

1.9.2 Bilanciamento in altezza

Un albero si dice **bilanciato in altezza** se le altezze dei sottoalberi sinitro e destro di ogni nodo differiscono al più di uno.

$$\beta \leq 1$$

Definizione: un albero AVL è un ABR bilanciato in altezza.

1.9.3 Inserimento e rimozione

Inserimenti e rimozioni richiedono di essere modificati per mantenere il bilanciamento dell'albero.
L'operazione fondamentale per ribilanciare l'albero è la **rotazione semplice**.

1.9.4 Rotazione a sinistra

Per effettuare questa rotazione prendo il nodo problematico e effettuo una rotazione scambiandolo con il successivo ed il nodo scambiato diventerà figlio destro mentre il figlio del nodo precendete diventerà figlio sinistro.

1.10 Alberi 2-3

Un albero 2-3 è un albero in cui:

- Tutti i percorsi radice-foglia hanno la stessa lunghezza
- Le foglie contengono le chiavi (e i dati da memorizzare) e sono ordinate da sinistra verso destra in ordine di chiave crescente.
- Ogni nodo interno (non folgia) v ha 2 o 3 figli e mantiene due informazioni
  - S[v], **chiave massima** nel sottoalbero sinitro (2 o 3 figli)
  - M[v], **chiave massima** nel sottoalbero centrale (3 figli)
- Distribuzione dei valori k delle chiavi nei sottoalberi:
  - Sinistro $k \leq S[v]$
  - Centro $S[v] < k \leq M[v]$
  - Destro $k > M[v]$

1.11 Tabelle Hash

Le **tabelle hash** hanno una implementazione basata su una chiave k e array. Per ottenere la chiave sono presenti diverse tecniche di calcolo.
Ricapitolando, per realizzare una tabella hash efficiente abbiamo bisogno di:

- Un vettore
- Una funzione has calcolabile velocemente e che garantisca una buona distribuzione delle chiavi nel vettore
- Un meccanisco per gestire le collisioni

1.11.1 Problema delle collisioni

Una funzione hash h si dice **perfetta** se è iniettiva:

$$\forall u, \in U : u \neq v \rightarrow h(u) \neq h(v)$$

Se le collisioni sono inevitabili, è necessario trovare un metodo che le minimizzi, distribuendo **uniformemente** le chaivi negli indici della tabella hash.

1.11.2 Funzioni hash

E' necessario fare una premessa; nelle funzioni hash è **sempre possibile** trasformare una chiave complessa in un numero, (conversione in binario).

1.11.3 Metodo dell'estrazione

Le caratteristiche di questo metodo sono:

- Usa solo una parte della chiave
- Si seleziona una sottosequenza di p bit, con $m = 2^p$
- Solitamente dalle posizioni centrali

esempio: Verranno prese le cifre centrali 101000

$$\text{bin}(\text{“beer”}) = 000010\ 000101\ 000101\ 010010$$

- **Vantaggi:** molto veloce da calcolare
- **Svantaggi** rischio collisioni più alto di altri metodi

1.11.4 Metodo della divisione

Basata sul resto della divisione per m:

- **Vantaggio:** molto veloce
- **Svantaggio:** Suscettibile a speficifi valori di m. Per risolvere questo problema bisogna scegliere m come numero primo non troppo vicino ad una potenza di 2.

Esempio:

$$m = 12, k = 100 \rightarrow h(k) = 4$$

1.11.5 Metodo della moltiplicazione

Basato sulla moltiplicazione e il resto del numero

1. Sia A una costante, $0 < A < 1$
2. Moltiplichiamo k per A e prendiamo la parte frazionaria
3. Moltiplichiamo quest'ultima per m e prendiamo la parte intera

Esempi:

$$m =, k = 3, A = 0.8 \rightarrow h(k) = 2$$
$$m = 1000, k = 123, A \approx 0.6180339887... \rightarrow h(k) = 18$$

- **Svantaggi:** lento (più lento del metodo di divisione)
- **Vantaggi** Il valore di m non è critico
- **Come scegliere A?** $A \approx (\sqrt{5} - 1)/2 = 0.61803...$ **(Knuth)**

1.11.6 Metodo della codifica algebrica

Metodo utilizzando dal compilatore java basato su espressioni algebriche:

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + ... + k_1 x + k_0) mod\ m$$
$$k = k_n k_{n-1} ... k_1 k_0$$

Dove $k_0, k_1...$ possono essere, ad esempio, i bit della **codifica binaria** di k, oppure i **codici ascii** dei singoli caratteri di k.
x è un valore **costante**.

- **Vantaggi:** dipende da tutti i bit/caratteri della chiave
- **Svantaggi:** n addizioni e $n * (n-1)/2$ prodotti

1.11.7 Problema delle collisioni

Attraverso questi metodi elencati precedentemente siamo riusciti a ridurre il numero di collisioni, ma senza eliminarle.

Per risolvere questo problema la **complessita computazionale potrebbe aumentare a** n, possono essere utilizzate le seguenti tecniche:

1. **Concatenamento**
2. **Indirizzamento aperto**

1.11.8 concatenamento

Nella tecnica di **concatenameto** gli elementi con lo stesso valore hash h vengono memorizzati in una lista concatenata (linked list).

Il **fattore di carico** è dato dal rapporto tra numero di elementi memorizzati e dimensioni della tabella.

La **complessità** del concatenamento è la seguente:

- insert: $\Theta(1)$
- search: $\Theta(n)$
- delete: $\Theta(n)$

1.11.9 indirizzamento aperto

L'idea è quella di memorizzare tutte le chiavi nella tabella stessa, ed ogni slot contiene una chiave oppure *null*.

Inserimento: se lo slot prescelto è utilizzato, si cerca uno slot alternativo.

Ricerca: si cerca nello slot prescelto, epoi negli slot alternativi fino a quando non si trova la chiave oppure *null*.

Vengono utilizzati diversi algoritmi di indirizzamento, per esempio i seguenti:

Ispezione lineare
Il primo elemento determina l'intera sequenza. In questo modo si ottengono **lunghe sottosequenze**.

$$h(k, i) = (h'(k) + 1)$$

Ispezione quadratica:
L'ispezione iniziale è in $h'(k)$, mentre le succesive hanno un offset che dipende da una **funzione quadratica nel numero di ispezione**.

$$h'(k) + c_1 i + c_2 i^2$$

Doppio hashing:
Formato da **due funzioni ausiliari** di cui la prima h_1 fornisce la prima ispezione, mentre h_2 fornisce l'offset delle succesive ispezioni.

$$h(k, i) = (h_1(k) + ih_2(k))$$

1.11.10 Conclusioni hash table

Usare funzioni hash $h(k)$ che producano valori il più possibile uniformemente distribuiti è molto importante perchè altrimenti potremmo arrivare ad una complessità computazionale pari a $O(n)$.

Problemi con hashing:

- Scarsa locality of reference (cache miss)
- In base all'implementazione è in genere difficile ottenere le chiavi in ordine
- Sebbene il costo medio per operazione sia basso, la singola operazione può risultare molto costosa, ad esempio se occorre ridimensionare la tabell ae redistribuire le chiavi.

2 Scelta degli algoritmi

A seconda delle operazioni da eseguire è necessario adattare diverse tecniche di implimentazione di un algoritmo.

2.0.1 Implementazione su un vettore ordinato

Questo tipo di ricerca ha un costo computazionale basso nel caso in cui volessimo **ricercare degli elementi**. Costi computazionali:

- Ricerca $O(log\, n)$
- Inserimento $O(n)$
- Eliminazione $O(n)$

2.0.2 Implementazione su liste concatenate non ordinate

Questo implementazione converrebbe utilizzarla nel caso in cui volessimo **aggiungere o eliminare degli elementi**. Costi computazionali:

- Ricerca $O(n)$
- Inserimento $O(1)$
- Eliminazione $O(n)$

2.0.3 Implementazione alberi ABR

Implementazione basata su alberi binari. Costi computazionali:

- Ricerca $O(h)$
- Inserimento $O(h)$
- Eliminazione $O(h)$

2.0.4 Implementazione alberi AVL

Implementazione basata su alberi binari a altezza equivalente. Costi computazionali:

- Ricerca $O(log\ n)$
- Inserimento $O(log\ n)$
- Eliminazione $O(log\ n)$

2.0.5 Implementazione alberi 2-3

Implementazione basata su alberi binari ordinati con chiavi massime. Costi computazionali:

- Ricerca $O(log\ n)$
- Inserimento $O(log\ n)$
- Eliminazione $O(log\ n)$

2.0.6 Hash table

Implementazione basata su array, dove l'elemento con chiave k è memorizzato nel $k - esimo$ "slot" dell'array.

- Ricerca caso medio $O(1)$ Ricerca caso pessimo $O(n)$
- Inserimento caso medio $O(1)$ Inserimento caso pessimo $O(n)$
- Eliminazione caso medio $O(1)$ Eliminazione caso pessimo $O(n)$

2.1 Riepilogo

	search		insert		delete	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array ordinato	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)
Lista non ordinata	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)
ABR	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)
Albero AVL	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Albero 2-3	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Tabella Hash	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)

3 Algoritmi di ordinamento

3.0.1 ordinamento in loco:

L'algoritmo permuta gli elementi direttamente nell'array originale, senza usare un altro array di appoggio.

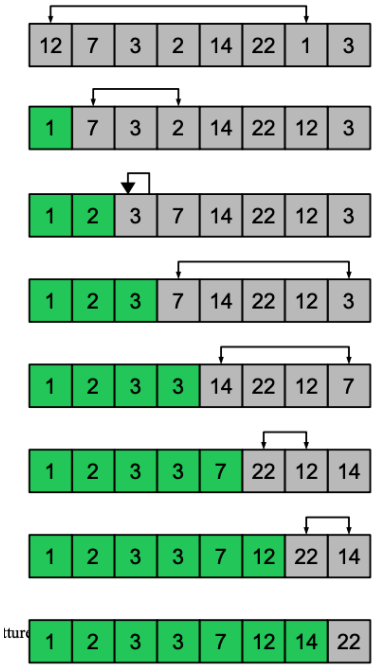
3.0.2 ordinamento stabile:

L'algoritmo preserva l'ordine con cui elementi con la stessa chiave compaiono nell'array originale.

3.1 Selection sort

Cerca il minimo in $A[k+1..n]$ e spostalo in posizione $k+1$.
La complessita di questo algoritmo è pari a:

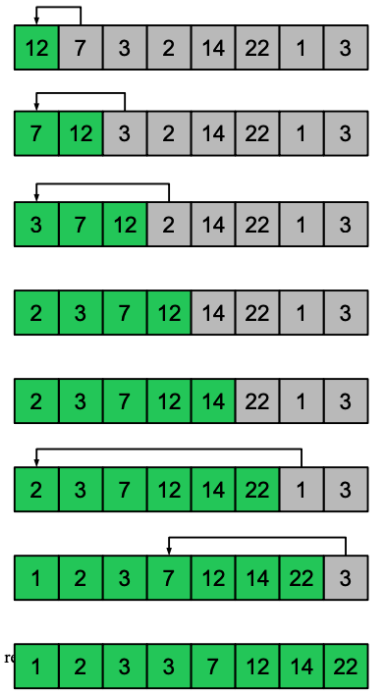
$$O(n^2)$$



3.2 Insertion sort

Inserisco l'elemento di posizione k+1 nella **posizione corretta** all'interno dei primi k elementi ordinati. Al termine del passo k, il vettore ha le prime k componenti ordinate.
Il costo computazionale di questo algoritmo è:

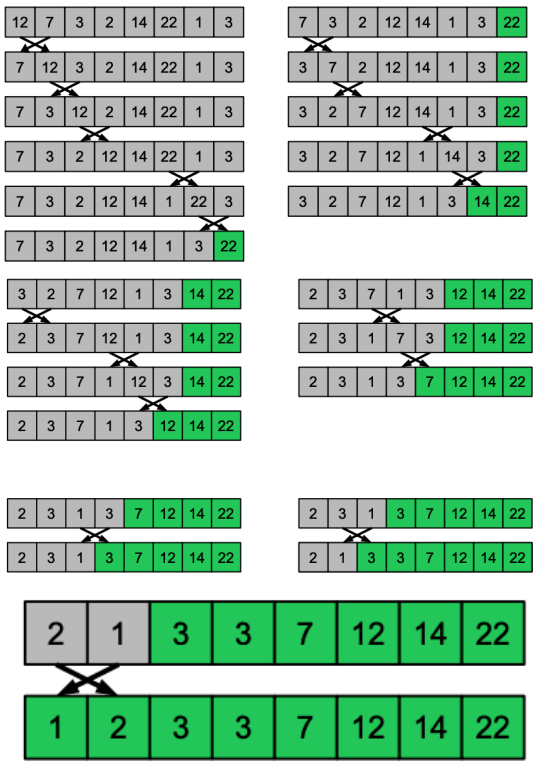
$$O(n^2)$$



3.3 Bubble sort

Ad ogni scansione scambia le coppie di elementi adiacenti che non sono nell'ordine corretto.
Ad ogni scansione **scambia le coppie di elementi adiacenti**. Dopo la prima scansione, l'elemento massimo occupa l'ultima posizione, dopo la k-esima scansione, i k elementi massimi occupano la posizione corretta in fondo all'array. Nel caso $pessimo - -ottimo$ bubble Sort ha costo:

$\Theta(n^2)$ $\Theta(n)$



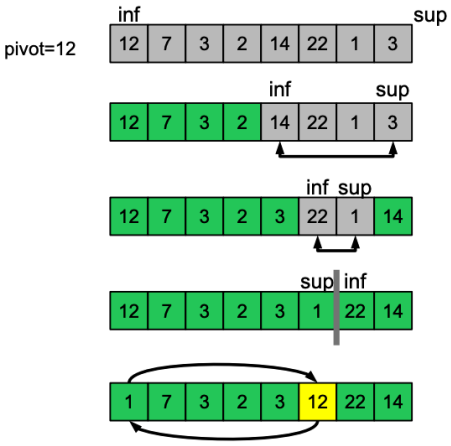
3.4 Quick sort

Scegli un elemento x del vettore v, e **partiziona il vettore in due parti** considerando gli elementi $\leq x$ e quelli $> x$
Ordina ricorsivamente le due parti.
Restituisci il risultato concatenando le due parti ordinate.

3.4.1 Partizionamento

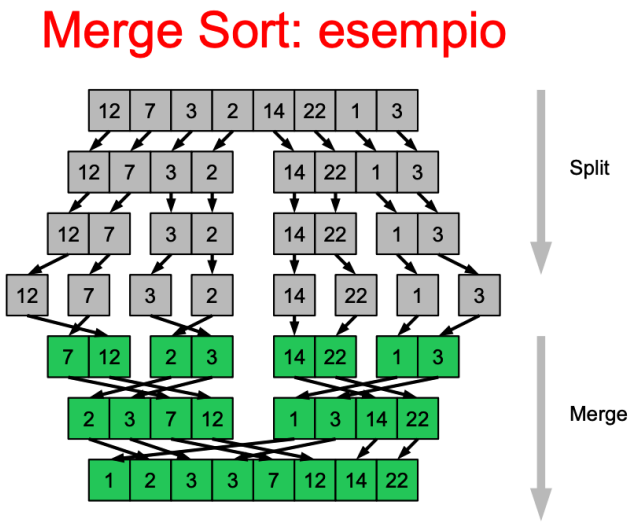
Manteniamo due indici, inf e sup, che vengono fatti **scorrere dalle estremità** del vettore verso il centro. Quando entrambi (inf e sup) non possono essere fatti avanzare verso il centro, si **scambia** $A[inf]$ e $A[sup]$.
Il costo quick sort: Dipende dal partizionamento:

caso peggiore: $\Theta(n^2)$ **caso migliore:** $\Theta(n \, log \, n)$



3.5 Merge Sort

Questo algoritmo **divide** $A[]$ in **due meta'** $A1[]$ e $A2[]$(senza permutare) di dimensioni uguali;
Applica **ricorsivamente** Merge Sort a $A1[]$ e $A2[]$.
Fonde (merge) gli array ordinati $A1[]$ e $A2[]$ per ottenere l'array $A[]$ ordinato.



3.6 Heapsort

Funzionamento:

1. Costruire un **max-heap** a partire dal vettore A[] originale, mediante l'operazione **heapify()**
2. Estrarre il **massimo** $(findMax() + deleteMax())$
3. **Inserire** il massimo in ultima posizione di A[].
4. **Ripetere** il punto 2. finché lo heap diventa vuoto

3.6.1 Albero binario perfetto

Un albero binario è **perfetto** se:

- Tutte le foglie hanno la stessa altezza h
- Nodi interni hanno grado 2

Un albero perfetto ha altezza $h \simeq log\ N$
Il numero di nodi è $N = nodi = 2^{h+1} - 1$

3.6.2 Albero binario completo

Un albero binario è **completo** se:

- Tutte le foglie hanno profondità h o h-1
- Tutti i nodi a livello h sono "accatastati" a sinistra
- Tutti i nodi interni hanno grado 2, eccetto al più uno

3.6.3 Max-heap

Un albero binario completo è un albero **max-heap** sse:

- Ad ogni nodo i viene associato un valore $A[i]$
- $A[Parent(i)] \geq A[i]$

3.6.4 Min-heap

Un albero binario completo è un albero **min-heap** sse:

- Ad ogni nodo i viene associato un valore $A[i]$
- $A[Parent(i)] \leq A[i]$

3.7 Operazioni su array heap

3.7.1 findMax()

Individua il valore massimo contenuto in uno heap.
Il massimo è sempre la radice, ossia A[1].
L'operazione ha costo $\Theta(1)$.

3.7.2 fixHeap()

Ripristinare la proprietà di max-heap.
Supponiamo di rimpiazzare la radice $A[1]$ di un max-heap con un valore qualsiasi, vogliamo fare in modo che $A[]$ diventi nuovamente uno heap.

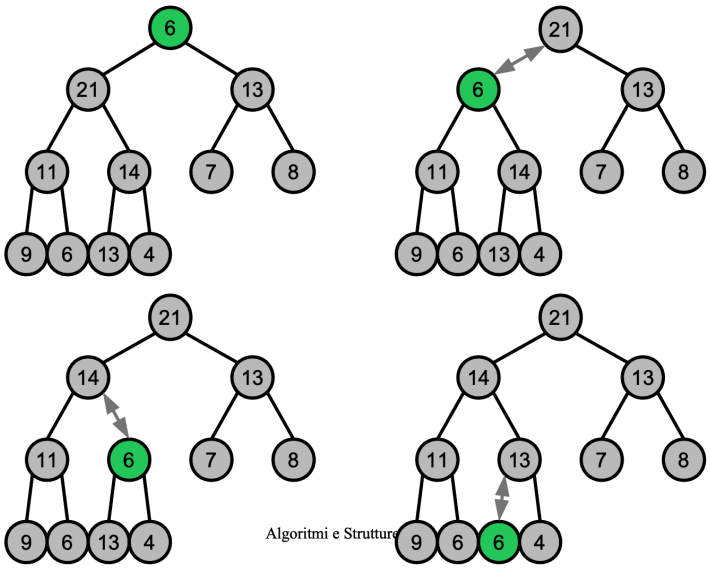
3.7.3 heapify()

Costruire uno heap a partire da un array privo di alcun ordine.

3.7.4 deleteMax()

Rimuovi l'elemento massimo da un maxheap $A[]$.

3.7.5 Esempio heapSort:



Algoritmi di ordinamento: sommario

- Abbiamo visto diversi algoritmi di ordinamento:
  - Selection Sort: ottimo/medio/pessimo $\Theta(n^2)$
  - Insertion Sort: ottimo/medio/pessimo $\Theta(n^2)$
  - Bubble Sort: ottimo $\Theta(n)$, (medio)/pessimo $\Theta(n^2)$
  - Quick Sort: ottimo $\Theta(n \log n)$,medio $O(n \log n)$,pessimo $\Theta(n^2)$
  - Merge Sort: ottimo/medio/pessimo $\Theta(n \log n)$ (non in-loco)
  - Heap Sort: ottimo/medio/pessimo $O(n \log n)$
- Nota:
  - Tutti questi algoritmi sono basati su confronti
    - le decisioni sull'ordinamento vengono prese in base al confronto (<,=,>) fra due valori

Esercizio: come modificare per avere caso ottimo $\Theta(n)$?

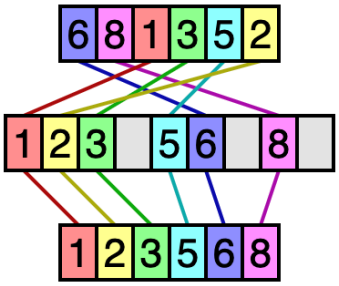
Esercizio: perché il caso medio è $\Theta(n^2)$?

4 Tecniche lineari di ordinamento

4.1 Counting Sort

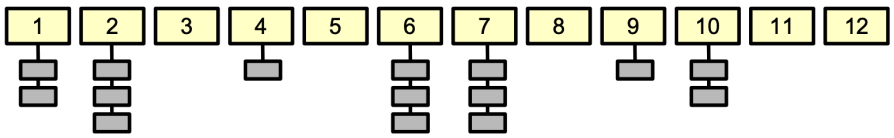
I valori di $A[0..n-1]$ appartengono all'intervallo $[0, k-1]$ (ciascun valore può comparire zero o più volte).
Costruisco un array $Y[0, k-1]$; $Y[i]$ conta il numero di volte in cui il valore i compare in $A[]$.
Ricolloco i valori così ottenuti in A.
Counting Sort: Costo

$$O(max\{n, k\}) = O(n + k) = O(n)$$



4.2 Bucket Sort

Cosa succede se i valori da ordinare non sono numeri interi, ma record associati ad una chiave?
Possiamo usare liste concatenate.

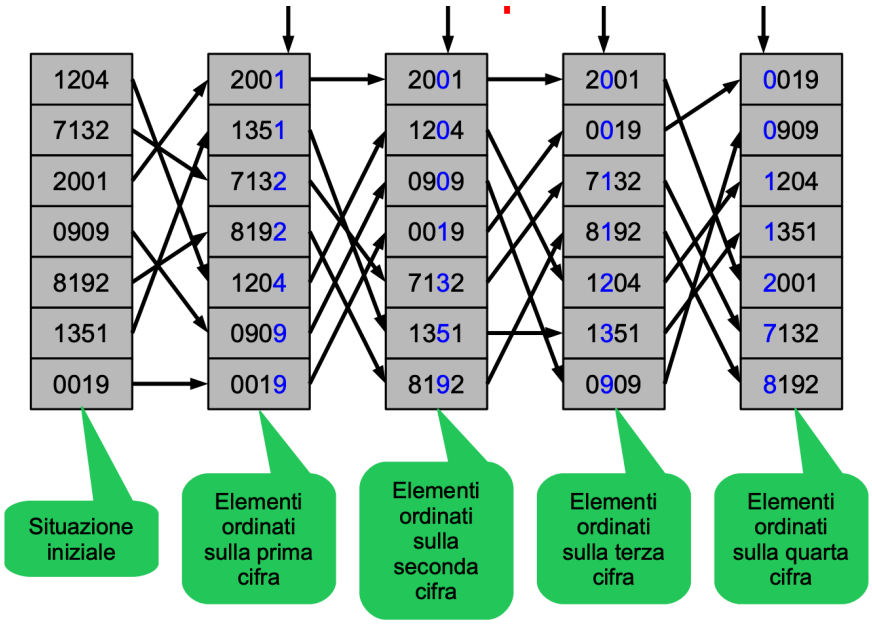


Bucket Sort: Costo

$$O(n+k)$$

4.3 Radix Sort

Supponiamo di voler ordinare n numeri con 4 cifre decimali.
Questo richiederebbe $n + 10000$ operazioni; se $n \, log \, n < n + 10000$, questo non sarebbe conveniente.
Prima ordino in base alla cifra delle **unità**.
Poi ordino in base alla cifra delle **decine**.
Poi ordino in base alla cifra delle **centinaia**.



5 Selezione del k-esimo

Consideriamo il seguente problema:
Selezione del k-esimo minimo: dato un array $A[1..n]$ di valori distinti e un valore $1 \leq k \leq n$, trovare l'elemento che è maggiore di esattamente $k-1$ elementi.
Mediano: il valore che occuperebbe la posizione $(n/2)$ se l'array fosse ordinato.
I **motori di ricerca** producono molti risultati a fronte di una singola query. I risultati vengono mostrati in pagine, in ordine decrescente di rilevanza. È **inutile ordinare tutti i risultati** in base alla rilevanza.
Verifichiamo ora i costi computazionali dei singoli casi:
Ricerca del minimo:

$$T(n) = n - 1 = \Theta(n)$$

Ricerca del secondo minimo:

$$T(n) = 2n - 3 = \Theta(n)$$

Selezione del k-esimo elemento:

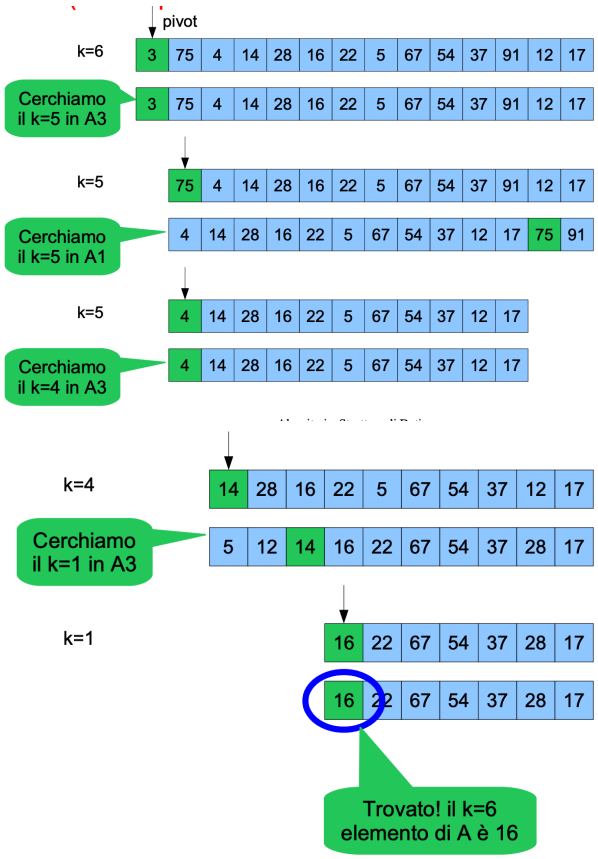
$$T(n) = \Theta(kn)$$

Selezione del valore mediano:

$$T(n) = O(n + k\ log\ n) = O(n + (n/2)log\ n) = O(n\ log\ n)$$

5.0.1 Adattamento di quicksort al problema della selezione:

In questo modo divido il mio array in più partizioni, andando a eliminare quelle inutili.



5.0.2 Analisi dell'algoritmo quickSelect()

Costo nel caso ottimo:

$$T(n) = T(n/2) + n = \Theta(n)$$

Costo nel caso pessimo:

$$T(n) = T(n-1) + n = \Theta(n^2)$$

Costo nel caso medio:

$$T(n) \leq 4n$$

6 Code con priorità

Le code con priorità sono strutture dati che mantengono il minimo (massimo) in un insieme dinamico di chiavi.

$$coda = key|elem$$

Sono presenti due possibili implementazioni:

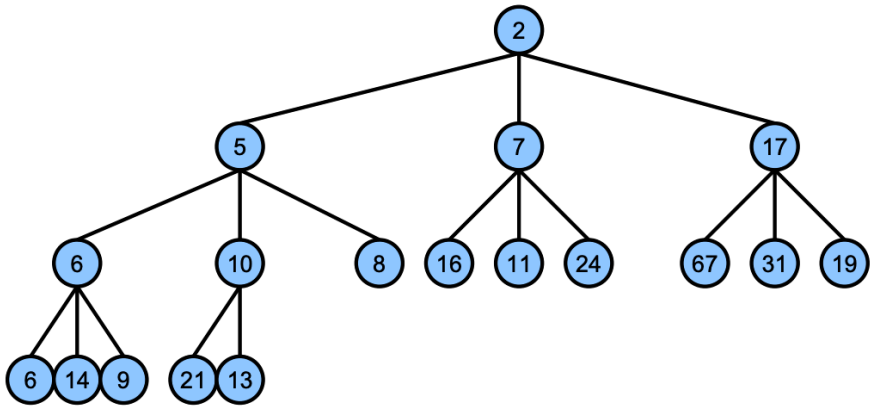
- d-heap
- heap binomiali e heap di fibonacci

6.0.1 d-heap

Un d-heap è un albero d-ario con le seguenti proprietà:

1. Un d-heap di altezza h è perfetto almeno fino alla profondità $h-1$; le foglie al livello h sono accatastate a sinistra.
2. Ciascun nodo v contiene una $chiave(v)$ e un elemento $elem(v)$. Le chiavi appartengono ad un dominio totalmente ordinato.
3. Ogni nodo diverso dalla radice ha chiave non inferiore ($\geq$) a quella del padre.

Esempio d-heap: $d = 3$



Un d-heap con n nodi ha **altezza** $O(log_d\, n)$

Riepilogo costi per d-heap

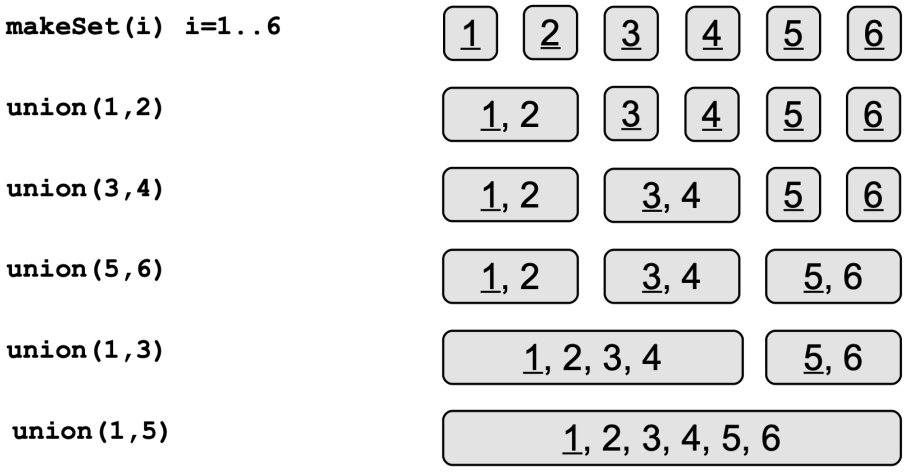
• findMin() → elem	O(1)
• insert(elem e, chiave k)	O(log$_d$ n)
• delete(elem e)	O(d log$_d$ n)
• deleteMin()	O(d log$_d$ n)
• increaseKey(elem e, chiave d)	O(d log$_d$ n)
• decreaseKey(elem e, chiave d)	O(log$_d$ n)

7 Union find

L'**union find** è una struttura dati basata sugli insiemi.
Essa può creare un insieme a partire da un singolo elemento, unire due insiemi, identificare l'insieme a cui appartiene un elemento. Gli insiemi contengono complessivamente $n \leq k$ elementi. Ogni insieme è identificato da un rappresentante univoco.
Operazioni Union find:

- $makeSet(elem\ x)$: Crea un insieme il cui unico elemento (e rappresentante) è x. Esso non deve appartenere ad un altro insieme esistente.
- $find(elem\ x) \rightarrow name$: Restituisce il rappresentante dell'unico insieme contenente x.
- $union(name\ x, name\ y)$: Unisce i due insiemi rappresentati da x e da y. Assumiamo che il nome del nuovo insieme sia x. I vecchi insiemi devono essere distrutti.

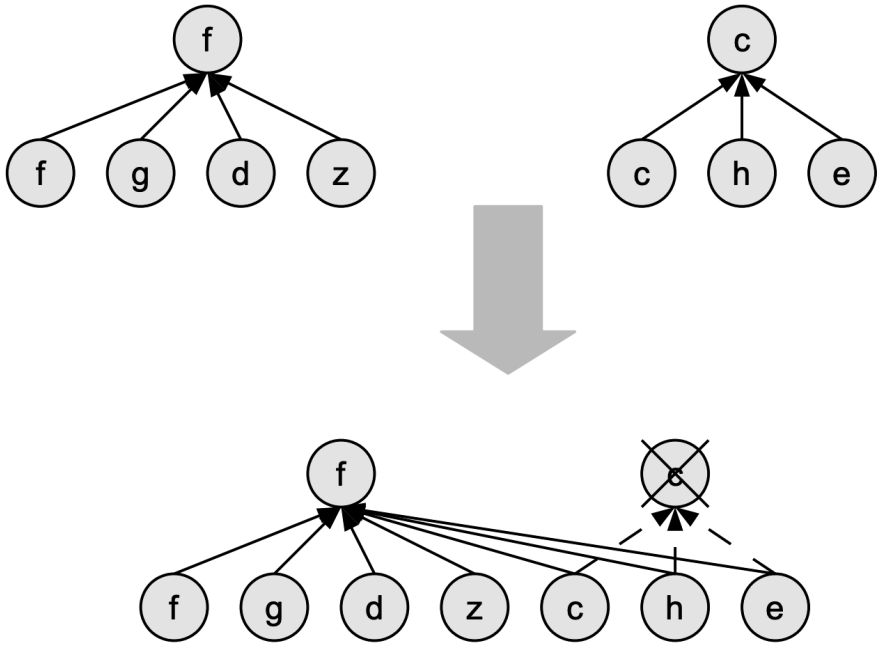
Esempio:



7.1 QuickFind

Ogni insieme viene rappresentato con un **albero** di altezza uno.

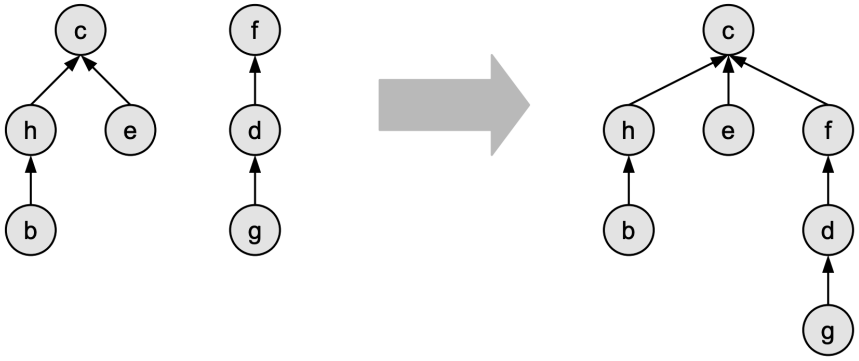
- Le foglie dell'albero contengono gli elementi dell'insieme.
- Il rappresentante è la radice.



7.2 QuickUnion

Implementazione basata su foresta.

- Si rappresenta ogni insieme tramite un albero radicato generico.
- Ogni nodo dell'albero contiene l'oggetto e il puntatore al padre.
- Il rappresentante è la radice.



Riepilogo

	QuickFind	QuickUnion
makeSet	O(1)	O(1)
union	O(n)	O(1)
find	O(1)	O(n)

E' conveniente utilizzare **QuickFind** quando le $union()$ sono rare e le $find()$ frequenti.
E' conveniente utilizzare **QuickUnion** quando le $find()$ sono rare e le $union()$ frequenti.

8 Divide et Impera

- Divide-et-impera
  - Un problema viene suddiviso in sotto-problemi, che vengono risolti ricorsivamente (top-down).
- Algoritmi greedy
  - Ad ogni passo si fa sempre la scelta che in quel momento appare ottima; le scelte fatte non vengono mai disfatte
- Programmazione dinamica
  - La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi

Tecnica divisa in **3 fasi** fondamentali:

1. **Divide:** Dividi il problema in sotto-problemi indipendenti, di dimensioni *minori*.
2. **Impera:** Risolvi i sotto-problemi ricorsivamente.
3. **Combina:** Unisci le soluzioni dei sottoproblemi per costruire la soluzione del problema di partenza

8.1 Algoritmi greedy

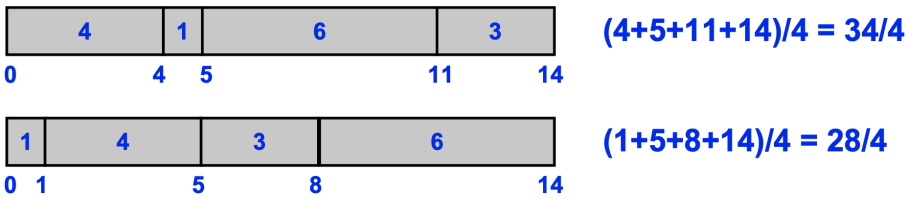
Quando applicare la tecnica greedy?

- Quando è possibile dimostrare che esiste una **scelta ingorda**.
  - Fra le molte scelte possibili, se ne può facilmente individuare una che porta sicuramente alla soluzione ottima.
- Quando il problema ha **sottostruttura ottima**.
  - "Fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale".

8.2 Algoritmo di scheduling

Algoritmo che si basa in base al tempo medio di eseguzione:

- 1 processore, n job p_1 , p_2, ..., p.
- Ogni job p_i ha un tempo di esecuzione $t[i]$.
- Minimizzare il tempo medio di completamento.



8.3 Codifica di Huffman

Questa codifica viene utilizzata per risolvere il problema di compressione, (compressione di un file). Viene utilizzata una tecnica detta **codifica di caratteri**

- Si usa una **funzione di codifica f:** $f(c) = x$
  1. c è un carattere preso da un alfabeto $\sum$
  2. x è una rappresentazione binaria del carattere c
  3. c è rappresenrtato da x in modo efficiente
- Una sequenza di caratteri $c_1c_2c_n$ viene codificato con la sequenza di bit $f(c_1)f(c_2)f(c_n)$
- data una qualsiasi codifica, deve essere sempre possibile decodificarla durante la lettura seuqenziale bit-dopo-bit.

Dobbiamo utilizzare una codifica che minimizza la dimensione del nostro file.

8.3.1 Codifica a lunghezza fissa

- Possibili car.: **'a'** **'b'** **'c'** **'d'** **'e'** **'f'**
- frequenze: **45%** **13%** **12%** **16%** **9%** **5%**

- Supponiamo di avere un file di n caratteri.
- Codifica tramite ASCII (8 bit per carattere).
- Codifica basata sull'alfabeto (3 bit per carattere).

8.3.2 Codifica a lunghezza variabile

- Caratteri: **'a'** **'b'** **'c'** **'d'** **'e'** **'f'**
- Codifica: **0** **101** **100** **111** **1101** **1100**
- Costo totale:
  (0.45*1+0.13*3+0.12*3+0.16*3+0.09*4+0.05*4)*n=2.24n

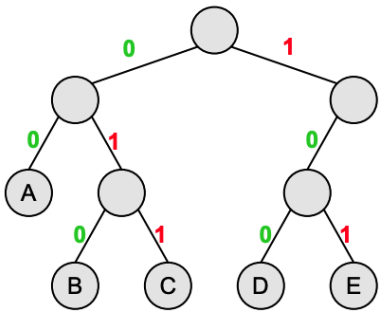
- Codifica a lunghezza variabile
- **codice a prefisso** (senza prefissi):
  - **nessun codice è un prefisso di un altro codice**.
  - Condizione richiesta per permettere sempre la decodifica durante la lettura bit-dopo-bit.

In questo modo avendo dei **prefissi univoci**, non appena troviamo una sequenza di bit possiamo risalire alla decodifica della parola. In questo modo **lettere frequenti** saranno composte da **pochi bit**.

8.3.3 Codici di Huffman

Rappresentazione del codice come un albero binario

- Figlio sinistro: 0 Figlio destro: 1
- Caratteri dell'alfabeto sulle foglie

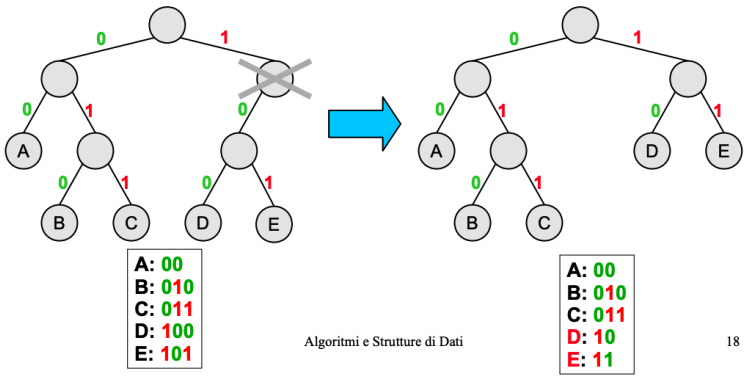


A: 00
B: 010
C: 011
D: 100
E: 101

Algoritmo di decodifica:

1. parti dalla radice
2. leggi un bit alla volta percorrendo l'albero:
   0: sinistra
   1: destra
3. stampa il carattere della foglia
4. torna a 1

Questo esempio per essere ottimizzato deve avere anche un figlio destro al primo livello di profondità.



Il **principio del codice di Huffman** è:

- **Minimizzare la lunghezza dei caratteri** che compaiono più frequentemente.
- Assegnare ai caratteri con la frequenza i codici corrispondenti ai percorsi più lunghi all'interno dell'albero.

1. Inizialmente costruiamo una lista ordinata di nodi, in cui ogni nodo continee un carattere e il numero di volte in cui quel carattere compare nel file.

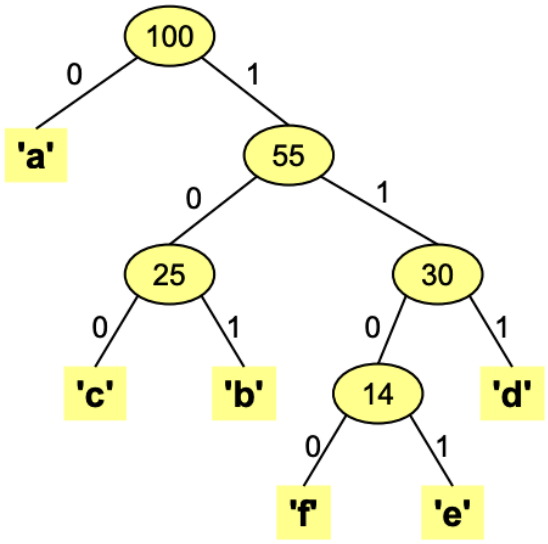
$$"f" : 5 \quad "e" : 9 \quad "c" : 12 \quad "b" : 13 \quad "d" : 16 \quad "a" : 45$$

2. Rimuovere i due nodi con frequenze minori.

3. Collegarli ad un nodo padre etichettato con la frequenza combinata (sommata).

$$"c" : 12 \quad "b" : 13 \quad "d" : 16 \quad "a" : 45$$

4. Aggiungere il nodo combinato alla lista, mantenendola ordinata in base alla frequenza.



Vantaggi:

- Semplici da programmare
- Solitamente efficienti
- Quando è possibile dimostrare la proprietà di scelta greedy danno la soluzione ottima

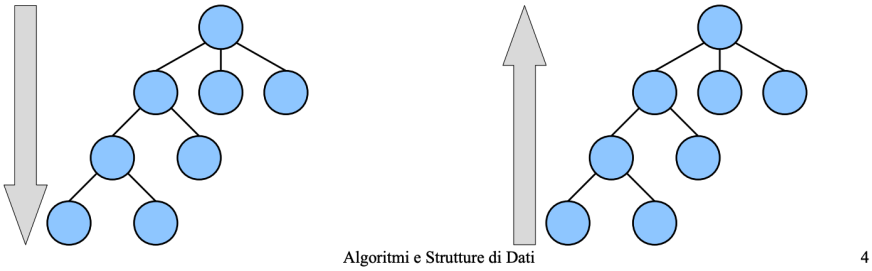
- La soluzione sub-ottima può essere accettabile

Svantaggi:

- Non tutti i problemi ammettono una soluzione greedy
- Quindi, in certi casi gli algoritmi greedy non possono essere usati se si vuole la soluzione ottima

9 Programmazione dinamica

- Divide-et-impera
  - Tecnica ricorsiva
  - Approccio top-down
  - Vantaggiosa quando i sottoproblemi sono indipendenti
- Programmazione dinamica
  - Tecnica iterativa
  - Approccio bottom-up
  - Vantaggiosa quando ci sono sottoproblemi ripetuti



Algoritmi e Strutture di Dati 4

1. **Sottostruttura ottimale**, deve essere possibile combinare le soluzioni dei sottoproblemi.
2. **Sottoproblemi ripetuti**, che ricompaiono constantemente.